NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

INVESTIGATION INTO EFFICIENT CONVERSION METHODS BETWEEN RESIDUE AND BINARY SYSTEMS

by

David E. Gilbert

September 1991

Thesis Advisor: Chyan Yang

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) EC	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
INVESTIGATION INTO EFFICIENT CONVERSION METHODS BETWEEN RESIDUE AND BINARY SYSTEMS

12. PERSONAL AUTHOR(S)
GILBERT, David E.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	September 1991	56

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	residue number systems; VLSI; ROM; programmable logic array

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Residue number systems (RNS) can efficiently perform addition, substraction, and multiplication in a parallel and fault tolerant manner. Because of this, they hold significant promise for use in digital signal processing, where high speed arithmetic operators are needed. However, the difficulties in using RNS, such as magnitude comparison between two RNS values, division, and determining overflow or underflow out of system range, have prevented more widespread use of these systems. This thesis investigates traditional methods to perform comparisons and to propose some new ones. Proposals include residue number system with quotient (RNS-Q), residue number system quotient-on-demand (RNS-QD), and pipelined conversions from traditional RNS to a mixed radix representation. These proposals will be compared with traditional methods with respect to silicon area needed for implementation, speed with which they can be developed, and VLSI techniques utilized to carry out the design.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL YANG, Chyan	22b. TELEPHONE (Include Area Code) 408-646-2266 22c OFFICE SYMBOL EC/Ya

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

UNCLASSIFIED

Approved for public release; distribution is unlimited

Investigation into Efficient Conversion Methods Between Residue and Binary Systems

by

David E. Gilbert
Lieutenant, USN
B.S.E.E., University of South Carolina, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

September, 1991

ABSTRACT

Residue number systems (RNS) can efficiently perform addition, subtraction, and multiplication in a parallel and fault tolerant manner. Because of this, they hold significant promise for use in digital signal processing, where high speed arithmetic operators are needed. However, the difficulties in using RNS, such as magnitude comparison between two RNS values, division, and determining overflow or underflow out of system range, have prevented more widespread use of these systems. This thesis investigates traditional methods to perform comparisons and to propose some new ones. Proposals include residue number system with quotient (RNS–Q), residue number system quotient–on–demand (RNS–QD), and pipelined conversions from traditional RNS to a mixed radix representation. These proposals will be compared with traditional methods with respect to silicon area needed for implementation, speed with which they can be developed, and VLSI techniques utilized to carry out the design.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. HISTORY 2
C. BASIC CONCEPTS 4
1. RNS 4
2. Moduli Set Choice 5
3. Chinese Remainder Theorem (CRT) 6
4. Mixed Radix Representation 7
5. Redundant Residue Number Systems 8
6. VLSI Overview 10
7. Programmable Logic Arrays (PLA) 11
D. THESIS OVERVIEW 12
II. ANALYSIS 14
A. NAIVE SOLUTION 14
B. PROPOSED ALTERNATIVE SOLUTIONS 16
1. RNS with Quotient 16
2. RNS with Quotient on Demand (RNS-QD) 19
3. Pipelined Mixed Radix Conversion 21
III. IMPLEMENTATION 23
A. GETTING STARTED 23
B. IMPLEMENTATION PROCESS 24
1. Initial Test 25
2. In-Depth Testing 27

C. DESIGN VERIFICATION 30
IV. CONCLUSIONS 32
APPENDIX A: C CODE UTILIZED 34
1. C Code for RNS to Binary Converter 34
2. C Code for Quotient Table Generation 36
APPENDIX B: SAMPLE EQUATIONS 38
1. Output from QLUGEN.C 38
2. Reduced Equations from *Espresso* 39
APPENDIX C: SAMPLE RNL FILES 40
1. Sample RNL Execution File 40
2. Sample RNL Simulation Output 41
REFERENCES 44
INITIAL DISTRIBUTION LIST 46

LIST OF TABLES

2.1 Illustration of RNS Implementations 18

3.1 Results of Minimization By Espresso 29

3.2 In Depth Test Results 30

LIST OF FIGURES

2.1 Example of a ROM Based RNS to Binary Converter [Ref. 17] 15

3.1 Comparison of RNS and RNS-QD PLA Implementation 26

ACKNOWLEDGMENT

I would like to thank the United States Navy for making this unique educational experience available to me, and all the staff and instructors who have made this a rewarding tour. Special thanks go to Dr. Yang and Dr. Butler for their help as advisors and friends. Lastly, but definitely not the least, I would like to thank my wife, Linda, for putting up with me on a daily basis.

I. INTRODUCTION

A. BACKGROUND

Residue Number Systems (RNS) have been investigated for quite some time for use in computer arithmetic implementations. There have been many stumbling blocks that have limited, or at times prevented, them from becoming more commonplace in computer systems. Frequently encountered difficulties are division, sign determination, detecting underflow and overflow, and comparing two RNS values. The primary focus of this thesis is the development of more efficient methods of comparing two residue numbers.

RNS representations lend themselves best to applications that require frequent addition, subtraction, and/or multiplication. These operations take advantage of the carry–free and parallel nature of residue arithmetics. Hence, they are ideally suited for signal processing techniques [Ref. 1]. They have also been shown to be of potential value in solving linear equations that are ill–conditioned [Ref. 2]. Researchers have done substantial work in the areas of optimizing arithmetic manipulations performed by RNS, but the basic means of comparison has involved a form of table look–up for a conventional weighted number value [Ref. 3, 4, 5]. Table look–ups tend to require large amounts of silicon area and are not very efficient in terms of speed of conversion, thereby making them the primary bottleneck preventing more widespread use of RNS.

This chapter includes the essential material needed for a basic understanding of modular/residue mathematics. Section C.6, in this chapter, provides some basic

VLSI design considerations and tradeoffs. It is not intended to be comprehensive, but should help one to understand the fundamentals.

B. HISTORY

Residue, or modulo, number systems have been identified since approximately 100 A.D. Their discovery has been jointly credited to China's Sun Tzu and the Greek mathematician Nichomachus [Ref. 2]. Credit seems to be most commonly given to Sun Tzu for a verse he wrote describing a three modulus RNS with prime moduli 3, 5, 7 repeated here:

> We have things of which we do not know the number
> If we count them by threes, the remainder is 2,
> If we count them by fives the remainder is 3,
> If we count them by sevens the remainder is 2,
> How many things are there?
> The answer, 23.

The rule stated in the verse has come to be known as the Chinese Remainder Theorem (CRT).

During the Ming Dynasty (1368AD – 1643AD) Hsin Tai–Wei may have published the first proof of the CRT in a verse entitled "Hun Hsin Tiang Bing" (Counting Soldiers). Hsin Tai–Wei's verse is as follows:

> Three men walk together, their chance of reaching seventy so slight.
> Among the five plum trees, twenty–one blossoms did they yield.
> Seven sons at midmonth, happily did reunite.
> Divide the sum by 105, the answer is revealed.

This verse reiterates the modulo system with relatively prime moduli {3, 5, 7} and possessing a dynamic range of 105 ($3 * 5 * 7$). Euler is credited with the first rigorous mathematical proof of RNS published in 1734. Gauss also published this theorem and the overall theory of residue numbers in the nineteenth century in his *Disquisitiones Arithmetical.* [Ref. 1]

Although the bulk of the theoretical development of residue number systems had been completed by the end of the nineteenth century, there had been little practical use found for them. With the advent of the electronic computer new interest was developed in RNS methods. There was a flurry of activity after World War II primarily focused on the error detecting capabilities of the system to make vacuum tube computers more reliable. Fault tolerant and error detection/correction research became less important when the transistor was invented, which dramatically increased computer reliability. Increased activity in the RNS field was noted in the 50's and 60's, as attempts were made to use RNS in general–purpose computing machines. The difficulties encountered in handling sign detection, division, and comparison made RNS implementation undesirable in these machines.

Digital signal processing began to emerge as a significant distinct field of research in the 1960's, separating it from general computing machines. Cheney designed a digital correlator that was based entirely on residue arithmetic in 1961 [Ref. 6]. Unfortunately, this development did not receive much attention and there was again a lapse in the intensity of research conducted on residue arithmetics. VLSI (very large scale integration) rapidly accelerated electronic development in the 1970's. New VLSI tools created new techniques for system design and gave rise to new problems for DSP researchers. Traditional methods employed for digital signal processing were not very modular nor parallel in nature. Modularity and parallelism are two key issues when considering a VLSI design implementation (discussed in a later section); these two

issues brought RNS research back to the surface again in an effort to take advantage of the modular and parallel characteristics that are inherent to residue arithmetics. VLSI issues bring us to the present day, where we are still in need of more efficient methods for comparing two residue numbers.

Throughout the history of residue arithmetic development there are many periods of inactivity. Periods of disuse have been driven by development of techniques that were faster than what could be implemented using RNS methods, and lack of totally efficient implementations of modulo systems. During the eighteenth and nineteenth centuries some of the giants in mathematics did extensive work on theoretical development. The list includes Euler and Gauss. Modern researchers have often stated that we may be rediscovering facts about RNS that have been lost in time[Ref. 1]. Knuth states,"Perhaps some day highly highly parallel computers will make simultaneous operations commonplace, so that modular arithmetic will be of significant importance in 'real–time' calculations when a quick answer to a single problem requiring high precision is needed."[Ref. 2] The days of parallel computing are becoming more and more commonplace. There is a need for more rapid calculations, especially in the field of digital signal processing, that can utilize the advantages of RNS methods, while suffering very little from any of the disadvantages.

C. BASIC CONCEPTS

1. RNS

Residue Number Systems are formed by selecting several relatively prime moduli. Relatively prime refers to the fact that none of the moduli have any common factors other than unity i.e., for two distinct moduli m_j and m_k are relatively prime if and only if

$$gcd(m_j, m_k) = 1. \tag{1.1}$$

There are a variety of methods available to select relatively prime moduli [Ref. 2]. We are not forced into having to look for special distinct primes of the Mersenne type or anything else that exotic [Ref. 7]. One easy way to derive a set of three relatively prime moduli is done by using $2^n - 1$, 2^n, and $2^n + 1$ as the set. Common sense can also be a big help in determining if a choice of a moduli set is relatively prime.

Another important fact about Residue Number Systems is that they form finite or Galois fields. This is significant in that it establishes their inherent error detecting capabilities. The set $S_m = \{0, 1, \ldots , m - 1\}$ together with modulo m addition and multiplication forms a finite algebra, denoted $\{S_m, +, *\}$. If m is prime, then the set $\{S_m, +, *\}$ forms a finite or Galois field. However if m is not prime, then $\{S_m, +, *\}$ is a finite ring $R(m)$. Fields are essentially a set of elements in which we can perform the simple arithmetic operations of addition, subtraction, multiplication, and division without leaving the set. A finite field is, in the most basic terms, a field with a finite number of elements [Ref. 8].

Finite rings have a much weaker structure than do fields. One of their weaknesses is that there is no multiplicative inverse for all ring elements and no generator exists to generate all the elements of the ring [Ref. 1]. Of special interest is a ring that is formed with a modulus choice of 2^n, Obviously a power of two modulus will not be prime, so all these implementations will be rings. An RNS implementation can benefit from the choice of an exact power–of–two moduli in that the representation is an the length of standard words in most computers. For a three moduli set, this makes the other two choices quite easy, in that 2^n, $2^n - 1$ and $2^n + 1$ are all relatively prime to each other.

2. Moduli Set Choice

The choice of the moduli is governed by the range of distinct values one wishes to represent. The range of the system is determined by the product of the

moduli.

$$Range = M = m_r * m_{r-1} * ... * m_2 * m_1 \tag{1.2}$$

A system with relatively prime moduli 3, 5, and 7 would have a dynamic range of 105, the product of the moduli. The implications of the dynamic range are that these are the total number of values that can be uniquely represented by a residue number system. A moduli set must be chosen such that numbers in the system in which it is to be used do not typically go out of this range. When an overflow (or underflow) occurs the resultant RNS representation is an alias of some other value, and cannot be differentiated from that value. An example of this for the {3, 5, 7} RNS is that the value of 106 for this system is the same as the value for 1, i.e. (1, 1, 1) and would be interpreted as the value one if a conversion is performed.

Typically RNS have been made up with 3 or 4 moduli, but this is in no way meant as a limitation. One helpful hint is that the largest moduli would best serve the overall system implementation if it is a direct power of two, as explained before. The choice of a power of two holds other advantages than just word length; there is research that shows modulo adders and multipliers can be implemented at significant savings in terms of area and also gain some speed advantages for direct power of two implementations [Ref. 9].

Basically the idea is to work indirectly on the 'residues' instead of directly on some larger integer value. By doing this we can reduce the storage requirements for intermediate results and take advantage of the rapid addition, subtraction, and multiplication of these residual values.

3. Chinese Remainder Theorem (CRT)

The Chinese Remainder Theorem (CRT) is the basic building block for all residue number system development. It is undoubtedly one of the oldest theorems

still in use today. Mathematically, the CRT can be restated by the following theorem [Ref. 2]:

Theorem 1 *Let $m_1, m_2, ..., m_r$ be positive integers which are relatively prime in pairs (as previously stated above). Let $M = m_1 * m_2 * \cdots * m_r$ and let $a, u_1, u_2, ...,$ and u_r be integers. Then, there is exactly one integer u, which satisfies the conditions $a \leq u < a + m$ and $u \equiv u_j modulo\ m_j$ for $1 \leq j \leq r$.*

The proof of Theorem 1 is as follows:

Proof 1 *If $u \equiv v(modulo\ m_j)$ for $1 \leq j \leq r$, then $u - v$ is a multiple of m_j for all j, so Equation 1.1 implies that $u - v$ is a multiple of $m = m_1 m_2 \ldots m_r$. This argument shows that there is at most one solution to $a \leq u < a + m$. As u runs through the m distinct values $a \leq u < a + m$, the r–tuples $(u\ mod\ m_1, \ldots, u\ mod\ m_r)$ must also run through m distinct values, since Theorem 1 has at most one solution. But there are exactly $m_1 m_2 \ldots m_r$ possible r–tuples $(v_1 \ldots v_r)$such that $0 \leq v_j < m_j$. Therefore each r–tuple must occur exactly once, and there must be some value of u for which $(u\ mod\ m_1, \ldots, u\ mod\ m_r) = (u_1, \ldots, u_r)$.*

The CRT is the starting point for all RNS work, although some other techniques have been tried.

4. Mixed Radix Representation

Another form of representation for RNS is called the mixed radix representation (or system). An advantage of this form of representation is that it is a weighted format such that comparisons may be performed without further conversion. One method for performing the conversion from RNS to a mixed radix form is described in the following equations.

$$v_1 = u_1(modulo\ m_1) = u_1 \tag{1.3}$$

$$v_2 = [(u_2 - v_1) * c_{12}](modulo\ m_2) \quad (1.4)$$

$$v_3 = [((u_3 - v_1) * c_{13} - v_2) * c_{23}](modulo\ m_3) \quad (1.5)$$

$$v_r = (\ldots((u_r - v_1) * c_{1r} - v_2) * c_{2r} - \cdots - v_{r-1}) * c_{(r-1)r}(modulo\ m_r) \quad (1.6)$$

$$U = v_r\, m_{r-1} \cdots m_1 + \cdots + v_3\, m_2\, m_1 + v_2\, m_1 + v_1 \quad (1.7)$$

These equations describe the conversion process where the u_i's are the original RNS representation values, the v_i's are the mixed radix values, and U is the fully converted value to some decimal or binary form. Calculating the conversion constants, c_{ij}'s is accomplished by Euler's equation.

$$c_{ij} m_i \equiv 1 \ \ (modulo\ m_j) \quad (1.8)$$

It is important to note again that the v_i form of the mixed radix is a weighted number and may be compared directly to another value. The format of equations 1.4-1.8 illustrates how each value v_i is dependent on the preceding value, v_{i-1}, and all earlier values of v's. Due to this cascading of dependency on previously calculated values, conversion into this type of representation lends itself well to a pipelined form of conversion. Full conversion to the value U is not required if only a comparison is desired and could be enabled or disabled as necessary. An example is that the mixed radix form of the numbers (using the moduli set {7, 5, 3}) 35 and 23 are (2, 1, 2) and (1, 2, 2) respectively, while in RNS they would be (0, 0, 2) and (2, 3, 2). Looking at the mixed radix form it is obvious which represented number is larger, but this is not true for the RNS representation; in fact the value for 23 "appears" to be larger than the value for 35. Investigation into more detail of the pipeline implementation will be done in Chapter III.

5. Redundant Residue Number Systems

Redundant residue number systems (RRNS) are defined as residue number systems with additional redundant moduli. A choice is made of n moduli, called the

nonredundant moduli as in any residue number system, with an additional r relatively prime *redundant moduli.* The extra r moduli are not considered in the calculation of the range M of the system. The system's legitimate range remains the product of the nonredundant moduli, as in the equation 1.9.

$$M = \prod_{i=1}^{n} m_i \tag{1.9}$$

The additional redundant terms form a product to define the illegitimate range as shown in Equation 1.10.

$$R = \prod_{i=n+1}^{n+r} m_i \tag{1.10}$$

The overall number of unique values that can be represented is indicated by

$$MR = \prod_{i=1}^{n+r} m_i \tag{1.11}$$

which includes the redundancy R.[Ref. 10]

The following is an example of how a number could be represented using a RRNS implementation. Using the familiar RNS with moduli {7, 5, 3} the decimal number 23 is represented by the three–tuple (2, 3, 2). Adding the relatively prime redundant moduli of {8, 11} results in two additional terms (7, 1). Putting it all together we have the five–tuple (2, 3, 2, 7, 1).

Using redundant moduli allows for greater error checking and correction capabilities, thereby making the overall system more fault tolerant. Watson and Hastings have done research on RRNS that detect any errors in the residues and correct one of them [Ref. 10]. There are also algorithms for burst error detection and correction available for RRNS implementation [Ref. 11]. This capability makes the implementation desirability of RNS methods even greater when a strong degree of fault tolerance is required. The RNS with quotient implementation is a form of a redundant system introduced in Chapter II, only it is not formed using extra moduli.

6. VLSI Overview

There are many tradeoffs to be considered in any engineering design process. Primary design considerations when undertaking a VLSI (very large scale integration) design are modularity, regularity, area, and development time. These are not the only items to consider, but they form the cornerstones for a good design approach and are in keeping with the spirit of VLSI.

Modularity is a concept that takes into account, to some extent, the ease of mobility of a functional block within a given overall circuit and also its value in other circuit implementations. If a VLSI module is "well formed" the interaction it undergoes with other parts of the circuit can be easily and succinctly characterized. A highly modular circuit can be thought of as a properly written software subroutine. The subroutine can be embedded or called by a variety of main programs or other subroutines and only depends on what variables are passed into it and what variables it must return to the calling program. Likewise, the calling program need not be concerned with the internal operations of the subroutine, only that it returns the desired result when needed. A poorly written subroutine would rely on global constants or variables within a given program, thus making it, in its present form, highly immobile and therefore not very modular. Modularity of subroutines is basically the same concept in VLSI design. Primary differences are that the interface in the circuit design is a physical boundary that must be connected vice the passing of variables in a subroutine.[Ref. 12]

Regularity in a VLSI design is important to both the speed of development and also to the modularity. Optimizing each and every functional element in a circuit may result in a significant savings in total silicon area used and may also yield the highest speed of operation. The drawback to this approach is the long design time and the lack of a guarantee of a modular circuit. Using standard cells that have been

set up in an on-line library and using an iterative process for interconnection of the cells to form a functional block generally leads to faster design time and a modular cell. This is where the cost function AT (area * time delay) must be considered when deciding on a full custom design or a high degree of regularity [Ref. 13].

Area and time considerations for a VLSI design are fairly closely interlocked as evidenced from the previous paragraphs. To accomplish a particular design in the least amount of space implies a full custom design with every circuit optimized. Implementation of a circuit in the least amount of time leads one to rely entirely on library cells for circuit realization. There is also the speed of the circuit to be considered; it must be fast enough to be compatible with the system that will be using it. If a full custom circuit enjoys mass production and must perform at the highest speeds possible, then the time required to develop it may be justified. However, if it is a limited production circuit without any serious speed requirements or it is a design prototype to investigate feasibility for implementation, then the quickest development time is preferable.

7. Programmable Logic Arrays (PLA)

PLAs are highly regular and modular VLSI structures. They are composed of an AND gate array followed by an OR gate array. The PLA is a subset of the ROM structure [Ref.14]. PLAs allow the VLSI designer to implement any combinatorial logic function that may be characterized by a sum of products (SOP). An example of a SOP is given in equation 1.12. below.

$$X = A\overline{B}C\overline{D} + \overline{A}BCD + \overline{A}\overline{B}C\overline{D} \tag{1.12}$$

One primary difference between a ROM and a PLA is that a ROM can implement any combinatorial function desired given the number of inputs and outputs. The PLA requires the function to be implemented must be expressed as a SOP. Advantages of

using a PLA over a ROM are that there may be a substantial savings in terms of silicon area used in the implementation of a PLA. Some ROM structure applications have many unused output functions. The ROM may be thought of as a post office that maintains mailboxes for all of the people on the mailing list but only has mail for a small portion of those addressees on any given day. The mail boxes are always there but only a few are used regularly. Using a ROM to realize 8 functions of 16 variables requires a 65,536 8–bit word structure [Ref. 15]. This same function may well be realized at substantial area savings using a PLA and SOP minimization techniques.

The facts outlined above concerning design modularity and regularity coupled with available tools for SOP minimization and PLA realization led to the choice of PLAs implementation which is investigated in this thesis. The tools used were *espresso*, *eqntott*, *mpla*, *magic* and *rnl*. These tools are all described in "Still More Works by the Original Artists"[Ref. 16]. After considering other approaches, this seemed to be the best to minimize the overall development cost and was in keeping with the regularity advantage of VLSI design.

D. THESIS OVERVIEW

This thesis concentrates on investigating traditional comparison methods and offering some alternative solutions to this problem. Background work has been presented to develop RNS basics. Chapter II investigates conventional comparison methods, proposes new techniques for conversion, and analyzes the different comparison techniques in order to determine the overall efficiency of each method. Comparison efficiency is driven by the speed at which this operation can be performed and by the overall savings offered by the modulo system as compared to current methods. Conversion is a limiting factor if the values being used require frequent manipulation in a weighted or conventional format, that makes the operations of division, scaling,

and comparison easier to perform. Chapter III concentrates on the implementation of the alternative methods that have been proposed. This includes the steps taken to realize the design in a VLSI layout and design verification. Conclusions and recommendations for future study are presented in Chapter IV.

II. ANALYSIS

The traditional conversion method was previously alluded to in Chapter I and is examined in more depth in this chapter. It is to be analyzed in terms of its good points and drawbacks. Proposed alternatives are introduced and analyzed as far as what benefits can be derived from their implementation. Comparisons are made between both the new alternative solutions and the conventional method. Design tradeoffs between the different methods are also discussed.

A. NAIVE SOLUTION

ROM table look–ups are the most commonly used method for converting an RNS value to some other form to facilitate comparison. The term "naive solution" is not meant to imply that designers are naive for choosing this solution to the comparison problem. It is used to suggest that this is the most straightforward solution and is simply implemented. The overall cost function has not been fully evaluated in considering this alternative, which lies at the root of any VLSI implementation. There are some valid reasons for utilizing this approach, and they will be examined.

Substantial drawbacks to ROM look–up tables lie in the fact that their size grows as the dynamic range of the moduli set being used, ROMs are relatively slow devices, and typical implementations require the use of multiple ROM tables. A multiple table look–up is given as an example in Figure 2.1 [Ref. 17]. Once the conversion has been accomplished, there is still the matter of performing the comparison, which is handled by traditional comparator circuitry. The circuitry in Figure 2.1 forms as an intermediate step the mixed radix representation of the residue number. This could be further optimized by imposing conditions so that when only a comparison is

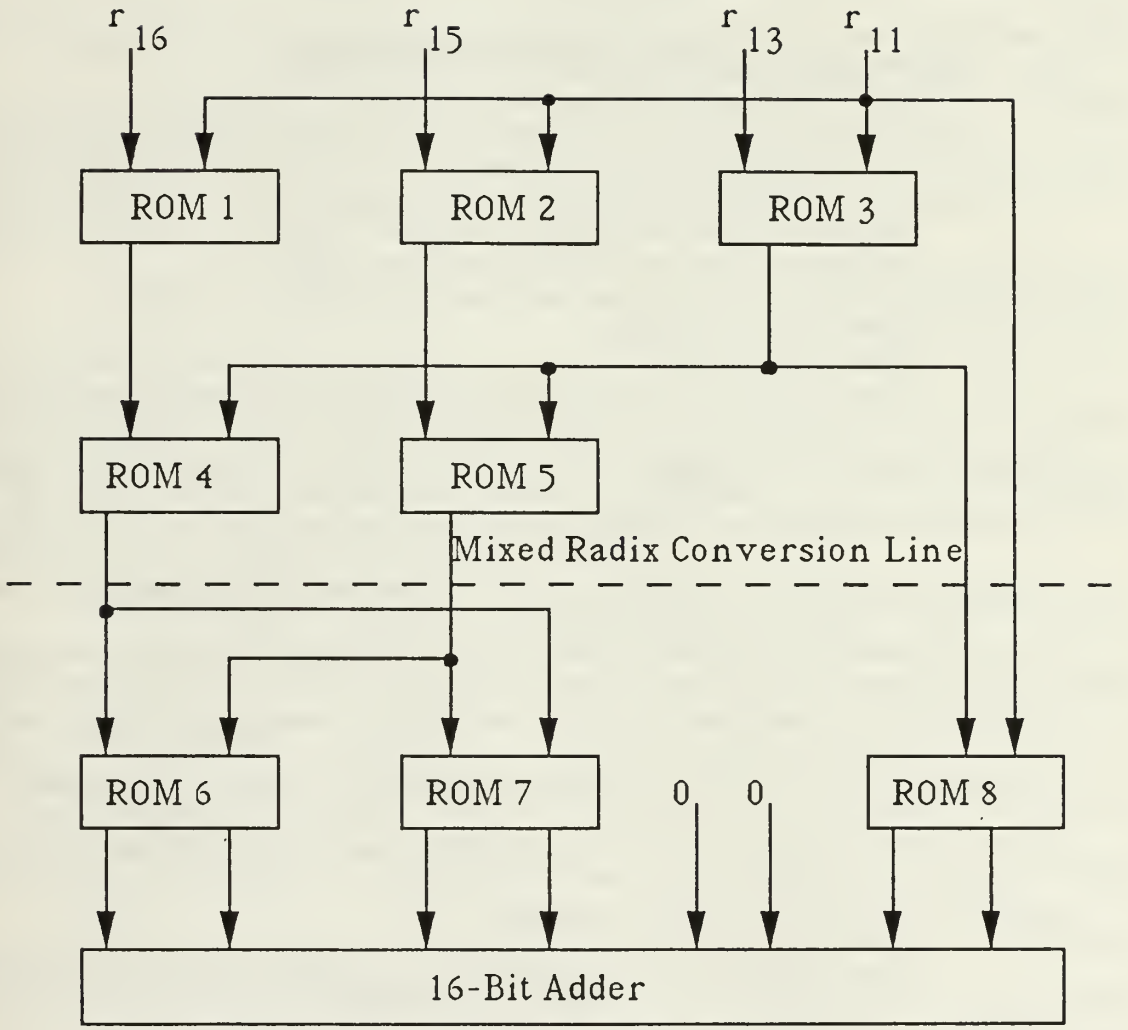

Figure 2.1: Example of a ROM Based RNS to Binary Converter [Ref. 17]

desired the mixed radix value could be utilized for the comparison and not wait on the full conversion. Doing this would reduce the time delay required for full conversion when actually not needed. Full conversion could be completed if the value is actually required for some external purpose.

B. PROPOSED ALTERNATIVE SOLUTIONS

The following three sections introduce alternatives to the "naive solution." Only one of them (the RNS–QD method) has been implemented and verified with a VLSI layout, the other two have been theoretically developed.

1. RNS with Quotient

This alternative uses the traditional RNS implementation with the additional element of the quotient, of the largest modulus, being part of the system. The name RNS with quotient (RNS–Q) comes from the fact that this quotient is now part of the representation. Advantages of this concept are that the quotient of any modulus and its residue form a unique value and are "ordered" in terms of magnitude, thus allowing direct comparisons to be accomplished. Conversions are only required with this system when a traditional value is required for output or some other use, but not to perform comparisons. The comparisons can be accomplished using traditional methods with "off the shelf" components.

The motivation for using the RNS–Q method is derived from Theorem 2. This theorem and its proof show the validity of such a representation.

Theorem 2 *Let* $M = m_r \ldots m_2 m_1$ *with* $m_r, \ldots, m_2$, *and* m_1 *all relatively prime. Then, the representation of* $\{i/m_j,\ i \bmod m_r, \ldots, i \bmod m_2, i \bmod m_1\}$ *is unique for any* $i \in [0,\ M-1]$.

Proof 2 *If* $a \neq b$ *and* a *and* $b \in [0, M-1]$ *using an* r *moduli system with moduli* $m_r, \ldots\ m_2$, *and* m_1, *with* $M = m_r \cdots m_2 m_1$. *The only parts of this in doubt are*

the a/m_j and b/m_j, because the rest of the representation are the traditional RNS and they are unique as a consequence of the CRT. If a/m_j and b/m_j are not unique representations, which they may well not be, the uniqueness of the remaining portion of the value preserves the uniqueness of the entire representation. QED.

RNS–Q representation may be thought of as the quotient serving as a redundant value, one that is not required to ensure uniqueness, but allows for immediate comparison without conversion. The RNS–Q system also forms a partition or an *equivalence class* of the set $[0, M-1]$ which is called the *quotient set* M/m_i [Ref. 18].

Drawbacks to this implementation are that extra bits are required to retain the quotient of the largest modulus, the system is no longer carry–free, and there is a potential loss of some of the fault tolerance of the traditional RNS. There will also be the extra burden of checking for overflow or underflow into or out of the largest moduli in order to update the quotient properly.

Loss of the carry–free characteristic poses the largest potential problem. Carry–free loss is due to the fact that whenever any arithmetic operation is performed on the RNS-Q numbers overflow out of, or underflow into, the largest residue must be reflected in the quotient also. With true RNS there is no requirement to check for the occurrence of overflow or underflow during normal operations. Traditional RNS does suffer from an aliasing problem when underflow or overflow occur. Failure to account for overflow and underflow will result in an invalid quotient. Overflow from the largest residue indicates that the quotient must be incremented by one, while underflow implies we must decrement the quotient once.

Table 2.1 outlines an RNS implementation using the relatively prime moduli set {2, 3, 5}. This system possesses a dynamic range of 30 values (0-29), and was chosen to illustrate some of the different representations that can be made. Shown

TABLE 2.1: Illustration of RNS Implementations

i	*RNS* $r_5 r_3 r_2$	*quotient* q_5	*RNS-Q* $q_5 r_5 r_3 r_2$	*Mixed Radix* $v_3 v_2 v_1$
0	0 0 0	0	0 0 0 0	0 0 0
1	1 1 1	0	0 1 1 1	0 0 1
2	2 2 0	0	0 2 2 0	0 1 0
3	3 0 1	0	0 3 0 1	0 1 1
4	4 1 0	0	0 4 1 0	0 2 0
5	0 2 1	1	1 0 2 1	0 2 1
6	1 0 0	1	1 1 0 0	1 0 0
7	2 1 1	1	1 2 1 1	1 0 1
8	3 2 0	1	1 3 2 0	1 1 0
9	4 0 1	1	1 4 0 1	1 1 1
10	0 1 0	2	2 0 1 0	1 2 0
11	1 2 1	2	2 1 2 1	1 2 1
12	2 0 0	2	2 2 0 0	2 0 0
13	3 1 1	2	2 3 1 1	2 0 1
14	4 2 0	2	2 4 2 0	2 1 0
15	0 0 1	3	3 0 0 1	2 1 1
16	1 1 0	3	3 1 1 0	2 2 0
17	2 2 1	3	3 2 2 1	2 2 1
18	3 0 0	3	3 3 0 0	3 0 0
19	4 1 1	3	3 4 1 1	3 0 1
20	0 2 0	4	4 0 2 0	3 1 0
21	1 0 1	4	4 1 0 1	3 1 1
22	2 1 0	4	4 2 1 0	3 2 0
23	3 2 1	4	4 3 2 1	3 2 1
24	4 0 0	4	4 4 0 0	4 0 0
25	0 1 1	5	5 0 1 1	4 0 1
26	1 2 0	5	5 1 2 0	4 1 0
27	2 0 1	5	5 2 0 1	4 1 1
28	3 1 0	5	5 3 1 0	4 2 0
29	4 2 1	5	5 4 2 1	4 2 1

are the traditional RNS, the quotients of the largest modulus, RNS–Q, and the mixed radix representation. It is easy to see that the traditional RNS does not yield a nicely weighted system of numeric values that we are accustomed to working with. Mixed radix and RNS–Q are also easily verified as ordered number systems and show that they may be compared without further manipulation.

It is obvious from Table 2.1 that {4, 2, 1, 0} is greater than {3, 3, 0, 0}. Our common sense would also tell us that {3, 0, 0} is greater than {2, 1, 0}, which is not true in the traditional RNS representation. The RNS–Q entity has the "look and feel" of of a traditional weighted decimal system (when ordered: q_5, r_5, r_3, r_2). This is because the quotient carries the most weight, just as a digit in the hundreds column of a decimal number carries more weight than a digit in the tens column.

The greatest advantage derived from an implementation of the RNS–Q system is that no conversion is necessary for comparison. Complete conversion can be accomplished by using a multiplier and an adder to multiply the quotient and its modulus and sum that result with the associated residue. RNS–Q systems were not implemented or tested. Implementation was not performed because an overall system would have to be developed to enable design tradeoffs between other methods.

2. RNS with Quotient on Demand (RNS–QD)

RNS–QD is based on the principles outlined for the RNS–Q representation given in the preceding section. The principle in RNS–QD is that the quotient is looked up when needed, hence the name RNS quotient on demand. Implementation of this system utilizes the advantages of the RNS–Q representation while maintaining the inherent strengths of the conventional RNS of being carry–free and fault tolerant. The loss of these traits were the principle drawbacks to the RNS–Q system.

Look–up tables were considered a disadvantage in the traditional RNS implementation. This was due to the fact that the table size grew in proportion to

the range of the system and were typically implemented in ROM–based structures. The fact that each RNS value was a unique representation made minimization of the number of sum–of–product terms required for implementation very slight. ROM–based table look–up systems also were a poor use of available silicon area, as described in the introduction, unless every memory site available is required. RNS–QD systems can be implemented so as to overcome some of these drawbacks.

PLA structures can be utilized to make more efficient use of available silicon real estate, than a similar realization based in ROMs, as long as the function can be expressed as a sum–of–products. This is true for both the RNS and RNS–QD systems. The advantage of the RNS–QD system is that there is substantial minimization that can be gained from the SOP terms used to define the function. Quotients derived from any of the moduli are not unique to each value in the range of the system. Each modulus' quotient set is determined by the product of the other moduli of the system and forms a partition of the set of all values in the range [Ref 17]. RNS with moduli set {3, 5, 7} has 15 different quotients (0 - 14) for the modulus seven, 35 quotients for three, and 21 quotients for five in the system range of 105. This also illustrates why selection of the largest modulus limits the number of carries required in a RNS–Q implementation.

Looking at the {3, 5, 7} system implementation we should choose 7 as the modulus of choice for which the quotient table will be generated. This means that there will be fifteen different quotients required to cover the range 0 - 104 ($3 * 5$). Ideally we could hope that the number of logic equations required to implement the table would reduce to fifteen. Although this is not the case, it is the number of output terms that we need to complete the quotient index.

3. Pipelined Mixed Radix Conversion

Fundamentally a pipeline in a computer system is formed when a large task is broken up along natural boundaries into sub–tasks. This is most frequently encountered in the execution of instructions within a central processing unit but, can also be found in floating point processors and other arithmetic devices. Breaking up a task into smaller units, called stages, allows for the clock cycle to be shortened such that all stage outputs are completed at the end of the cycle. Minimum clock cycle duration is limited to the slowest stage's speed. Cascading the stages together forms what is referred to as a pipe. Generally speaking the latency from entering the pipe to exiting the pipe is longer than if the task were not subdivided. The idea is that as long as there is one instruction entering the pipe every clock cycle theoretically, over long periods of time, instruction execution will approach one per cycle. [Ref. 19]

Conversion from RNS to mixed radix is an easy task, but the following example should clear up any doubts and also illustrate the cascading nature of the conversion. The first step is to calculate the conversion constants c_{ij} for the moduli set being used. Conversion constants are obtained using Equation 1.3, and, once they have been calculated for a given moduli, set they never have to be recalculated. Using the moduli set {5, 3, 2} (with $m_3 = 5$, $m_2 = 3$, and $m_1 = 2$), the conversion constants are: $c_{12} = 2$, $c_{13} = 3$, and $c_{23} = 2$. Proceeding with the rest of the conversion the RNS value (4, 1, 1) was chosen to be converted. Using Equations 1.4 through 1.7 the values of the v_i's can be obtained from the u_i's as follows:

$$v_1 = u_1 \; mod \; m_1 = u_1 = 1$$

$$v_2 = (u_2 - v_1)c_{12} \; mod \; m_2 = [(1 - 1) * 2] \; mod \; 3 = 0$$

$$v_3 = [((u_3 - v_1)c_{13} - v_2) * c_{23}] mod \; m_3 = [((4 - 1) * 3 - 0) * 2] mod 5 = 3$$

The mixed radix representation for the RNS value (4, 1, 1) is (3, 0, 1). A program can easily be written to perform this function automatically given the input moduli set.

Mixed radix conversion lends itself well to this type of implementation, due to the nature of the conversion each step depends in order on all the preceding steps. This is evidenced from equations 1.3 to 1.7 in Chapter I. Systems that perform large numbers of conversions or that perform many in bursts can benefit from pipelined structures. There are several stages that can be cascaded to form the pipe and the length of the pipe is in direct proportion to the number of moduli in the system. Individual stages are also fairly modular and can be built with good regularity, which lend themselves to the VLSI environment. The final stage, needed for full conversion, is not always required if only a comparison is to be done. Enabling or disabling the final stage could be done to reduce pipeline latency for situations when full conversion is not required.

III. IMPLEMENTATION

Circuitry for the RNS-QD system and the straightforward full conversion look-up tables were designed and implemented using programmable logic arrays. The following sections provide a detailed description of the design process.

A. GETTING STARTED

The first steps in any engineering design process are choosing the methods of implementation that best fit the task. This consisted initially of deciding how best to generate differing layouts for the purpose of determining which was more efficient. Generating layouts in VLSI can be extremely time consuming if there is a desire for a full custom realization. The decision was made to implement the traditional RNS with full conversion table look-up and the RNS-QD with a quotient look-up table utilizing PLAs. This was based on the fact that these designs could be easily accomplished, were very modular, and could be easily contrasted as to which was more efficient. RNS-Q and the pipelined MRC systems are of the type that their efficiency would be best demonstrated in a full system implementation and are not easily compared to the other two solutions.

To realize a PLA design, one must develop the logic equations necessary, minimize the initial equation set, and generate the PLA layout. Several design tools are available for aiding the designer. *Espresso* is a design tool for logic equation minimization that is part of the *magic* VLSI computer assisted design (CAD) tool [Ref. 16]. Utilization of *espresso* greatly reduces the development time required of the designer and in some cases makes an impossible problem reasonable. The output generated by *espresso* is formatted for direct use by the tool *mpla*. *Mpla* generates a PLA layout

automatically when given the SOP equations that must be realized. Without a tool such as *mpla*, the designer could proceed with a PLA design by laying out cells for basic AND and OR gates. The AND and OR gate cells could be connected together in the proper sequence so as to realize the same logic equation as done by *mpla*, but this is still more time consuming. There was only one tool missing - a tool to generate the logic equations was needed. The programming environment chosen was C. This choice was based on the need for rapid prototyping and the simplicity of the programs that were required.

The first level program written was to verify the structures of RNS and in the hopes of providing some additional insight into the interplay between the relatively prime moduli of the system. Further refinement of this entry level program resulted in versions that created RNS–Q type systems for graphic verification of the uniqueness theorem introduced in Chapter II. Another program modification yielded mixed radix representations of RNS, which allowed illustration of the fact that this is an "ordered" system. Final refinement resulted in two programs to generate logic equations in the output format required for use by *espresso*, the VLSI minimization tool to be used. One program generates equations for a traditional RNS table look–up conversion (cnvrtres.c), the other for RNS–QD quotient look–up table (qlugen.c), both of which are contained in Appendix A.

B. IMPLEMENTATION PROCESS

Once the programs were performing properly to enable generation of the logic equations, the question of what systems should be built arose. The decision was made to test several implementations that had similar dynamic ranges. This enabled checking of the hypothesis that by choosing a much larger, largest prime modulus, i.e. significantly larger than the other moduli, if there were any savings gained by

having less quotients in the table. The direct power of two moduli was also checked to investigate savings that had been realized for power of two adders and multipliers [Ref. 9]. Obviously one pitfall to this approach is that the dynamic ranges for two differing residue number systems will not be identical, but they can be close enough. The choice of a dynamic range is based on the fact that it is large enough so that there will be little chance of overflow out of the range. Dynamic range requirements are a floor setting, or minimum setting, much the same as number of bits needed in a computer. If a minimum word length needed is eight bits but you have a 16 bit machine there are advantages to using the larger word size computer, while placing no limitations on the requirements.

1. Initial Test

RNS sets that were initially evaluated were based on the requirements for a minimum dynamic range of 105. This allowed for use of the previously referred to system of relatively prime moduli {3, 5, 7}. Two other systems were chosen for comparison that had a dynamic range greater than 105. These two systems have moduli of {3, 5, 8} and {2, 5, 11} with dynamic ranges of 120 and 110 respectively. The assumption on dynamic range for this implementation is that the system requires a minimum range of 100. Another point of interest between these three systems is that they all maintain the same number of input (8) and output (4) bits. Maintaining an equal number of I/O bits is not necessarily a requirement, but provides for stability between the systems and, as a result, did not become a point of contention when the final evaluation was made. Implementation of these systems was done for both the traditional RNS and the RNS-QD methods to allow contrasting the costs of realization.

The implementation was realized using PLAs generated from the minimized logic equations from *espresso* and were fed directly to *mpla* for VLSI layout.

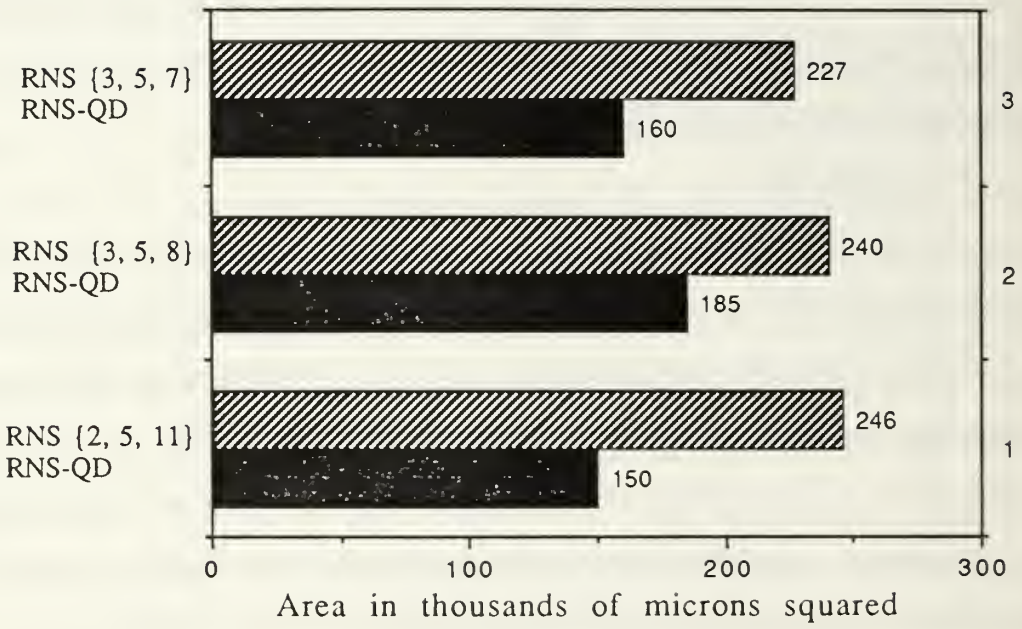

Figure 3.1: Comparison of RNS and RNS–QD PLA Implementation

Minimization by *espresso* resulted in an average reduction of 20 - 30% of the original logic equations generated by the C program for the RNS–QD system. Virtually no minimization was realized when *espresso* was used on the logic equations for the traditional RNS conversion look–up table. This is not surprising when one thinks about it. The RNS values are unique, and they correspond to unique integer numbers, so one should not expect to be able to realize any reduction in size. An exception was discovered when a direct power of two modulus was implemented (except where the modulus was 2 itself). In this case, there was some logic minimization encountered, but it was still less than what was afforded one in the quotient look–up table. This minimization contributed to reducing the width of the PLA, which in essence is reducing the number of terms that must be fed to the OR gate plane. The majority of the minimization that occurred in the quotient look–up table accounted for reductions in the height of the PLA, or the number of AND gates required for realization. There was also some reduction in the width of these structures for the RNS–QD format. Reduction in the height of the PLA has the greatest effect on the overall area, and the slight reduction in the width for the direct power of two conversion PLA still had an overall growth in total area from the previous smaller implementation. Figure 3.1 shows the differences in overall area of the RNS versus the RNS–QD implementations. The average savings in area gained by using the RNS–QD system is 30%.

2. In–Depth Testing

After performing this initial test there was a desire to attempt a more interesting problem. The University of Florida has been working on a pipelined mixed radix converter for the moduli set {101, 109, 113}, and it would be interesting to compare results [Ref.20]. Using the same C programs used to develop the logic equations seemed easily accomplished. Generating the complete set consumed all the available disk space and thereby shut down the ECE Vax. Looking at the *espresso* program,

also seems to indicate that a maximum of 3000 stack pushes are available before an error is received. This would have been exceeded by an attempt at minimization of the logic equations for this system, which are a little over one million.

To investigate some moduli sets with larger dynamic ranges, and to avoid computation problems, another set of moduli was chosen of approximately ten times the dynamic range of the first set. The sets of relatively prime moduli are {3, 5, 64}, {7, 11, 16}, and {3, 11, 32}. Use of these moduli sets gives a dynamic range of 960 - 1232. The choice of these sets was driven by the desire to further investigate the ideas of the "much larger moduli" hypothesis for savings in area, and the power of two savings that had been realized in adder and multiplier implementations [Ref. 9].

This time the generation of the logic equations was accomplished without putting any heavy strain on the Vax's available disk space. Minimization by *espresso* was fairly time consuming but was completed in about two hours for each set of moduli. Full PLA implementation was not required as from previous results analysis of the reduction in the amount of logic equations would be a good indicator of the size of each PLA. The results are tabulated in Table 3.1. From Table 3.1, it is easy to see that the most significant reduction came from the implementation of the set {3, 11, 32}. Earlier work would have lead us to believe, or at least hope for, the greatest reduction in the set {3, 5, 64}. What Table 3.1 doesn't show is that the number of output lines for the {3, 5, 64} set is four, while there are six output lines required for implementation of the {3, 11, 32} representation. Input and output lines had remained constant throughout the initial test set and were not variables to be evaluated. Decreasing the number of routing lines required is always welcome, and may be more significant than the extra area consumed by the larger PLA. The PLA may even be smaller due to this decreased bandwidth needed for the system with the largest quotient modulus, thereby needing less quotients as outputs.

TABLE 3.1: Results of Minimization By Espresso

Moduli Sets	# of Original Equations & Dynamic Range	# of Reduced Equations	Percent Reduction
{ 7, 11, 16}	1232	828	33%
{3, 11, 32}	1056	511	52%
{3, 5, 64}	960	555	42%
{3, 13, 29}	1131	831	26%
{3, 11, 31}	1023	556	46%
{3, 11, 34}	1122	556	50%

The actual PLA implementation was done by sending the reduced equation set to the VLSI tool *mpla* for automatic realization. The results of this step are contained in Table 3.2. The dimensions used for height, width, and area are in terms of the technology being used in microns. Three micron technology is being used, so the dimensions would be divided by three to obtain the actual dimensions in micrometers (or by nine micrometers for area), to get the size using one micron technology. Table 3.2 shows that the {3, 11, 32} implementation uses the least amount of area, as predicted by the least number of logic equations required to realize this system. The {3, 5, 64} system is only slightly larger than the {3, 11, 32} system and is not as wide. Width, of a PLA, is an indication of the number of gate delays from input to output. Although timing analysis was not performed on these circuits, the less gate transitions required implies the faster the operation of the circuit.

Testing was performed to investigate for power–of–two advantages that had been discovered previously [Ref. 9]. The implementations tested were for moduli sets {3, 11, 29}, {3, 11, 31}, {3, 11, 34}, and the previously implemented system {3, 11, 32}. The dynamic ranges, percent minimization, and area cost statistics are shown in Tables 3.1 and 3.2. Once again the system {3, 11, 32} holds an advantage

TABLE 3.2: In Depth Test Results

Moduli Sets	Height	Width	Total Area
{7, 11, 16}	7068	462	3265416
{3, 11, 32}	4388	390	1711320
{3, 5, 64}	4762	382	1819084
{3, 13, 29}	7092	438	3106296
{3, 11, 31}	4794	382	1831308
{3, 11, 34}	4770	398	1898460

over the other implementations, but the {3, 11, 31} system is a close second. Full implementation of the {3, 11, 31} system will require much more total area, because the residue multipliers and adders are much larger [Ref. 9]. It appears that there is fairly linear growth in the size of the PLA structure away from a direct power–of–two implementation, in both the positive and minus directions. The choice in this case would be the system with moduli set {3, 11, 32}, for the best overall savings.

C. DESIGN VERIFICATION

To completely authenticate a design there must be some type of verification performed to guarantee its validity. Verification for the implementation of the RNS–QD and traditional RNS PLA layouts was done with the use of a tool called *RNL* [Ref. 21]. *RNL* is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses simple resistance–capacitance model of the circuit that has been extracted from the VLSI layout done in *magic*. *RNL* allows for verification of the device of interest by using timing files and node transitions as inputs. There are other simulation tools that can be used if a more detailed circuit analysis is desired, but the goal of this testing is to verify that the for a given set of input vectors the proper outputs are received.

Design verification was performed on the first few PLAs implemented. A sample RNL file is contained in Appendix C, that includes timing, clock speeds, inputs, and outputs received for the system {3, 5, 7}. After ensuring that the logic equation generating programs functioned properly further design verification was not performed. Worst case timing analysis was not accomplished, but could also be done using RNL and the glitch detector [Ref. 21].

IV. CONCLUSIONS

Investigation into the usefulness of RNS in computer arithmetics has been going on for some time. One of the principle drawbacks has been the difficulty in comparing two residue numbers without conversion to some other form. The traditional method for comparison is a ROM based table look–up; which is relatively slow and uses a large amount area. Proposals for the use of RNS–Q, RNS–QD, and pipelined MRC have been presented and analyzed in this paper. These proposals offer savings in area required and may offer speed advantages.

The PLA implementation of RNS–QD offers a significant savings in terms of silicon area over the straightforward ROM look–up method. The larger moduli concept did not show an overall decrease in the size of the PLA required for implementation, but did yield a lower number of output lines, thereby reducing routing requirements for the circuit. Power–of–two investigation showed that there are some savings to be gained from implementing these systems, but was not as dramatic as previously discovered for the power–of–two adders and multipliers. There is still the necessity for a multiplier and an adder to facilitate full conversion to a conventional weighted number. If the need for comparisons occur much more frequently than full conversions, the PLA RNS–QD is a more viable method. However, if full conversions are required in conjunction with virtually every comparison, than this method may not yield a significant speed advantage over the ROM approach.

RNS–Q systems offer distinct advantages over both the traditional ROM and RNS–QD methods in terms of silicon real estate. This is especially true if the number of comparisons that a given system requires is very high compared to all other mathematical operations. The drawbacks to this approach are that the system is no

longer carry–free and because of this there is some loss in the inherent fault tolerance of conventional RNS. Loss of fault tolerance is most severe, if damage were to occur to the most significant (largest) modulus of the system, which would essentially disable the entire circuit.

The choice of the method to be employed must be approached from the view of what type of system is going to be using it to derive the best advantage. Each of the methods discussed have their own strong points. Considerations as to what is more crucial to the system, such as area occupied or speed of operation, will help to choose the implementation that will best fit the needs of the design.

APPENDIX A: C CODE UTILIZED

Enclosed in this appendix are the two C programs used to generate logic equations utilized for PLA generation.

1. C Code for RNS to Binary Converter

```
/*************************************************************
*       PROGRAM:        cnvrtres.c                          *
*       FUNCTION:       Generates logic equations in the    *
*                       format necessary to be used by the  *
*                       VLSI function "espresso". The output*
*                       derived can be used by "mpla" to    *
*                       create a PLA layout for a RNS to    *
*                       conventional binary value converter.*
*       AUTHOR:         David E. Gilbert                    *
*       VERSION:        1.2                                 *
*       DATE (last mod):29 AUG 1991                         *
*************************************************************/

#include <stdio.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

main(argc, argv)
int argc;
char **argv;
{
int mi[5], M, i, j, k, ctr[5], DONE, tmp;
int digit, tmp_div, raise_it;

M=1;
for(i=1; i< argc; i++) { mi[i]=atoi(argv[i]);
                         M = M*mi[i];
                       }
mi[4] = M;

/* CALCULATE THE POWERS OF 2 NEEDED FOR NUMBERS OF BITS NEEDED */

for(i = 1; i <= 4; i++)
  {
ctr[i] = 0;
tmp = mi[i];
```

```
DONE = FALSE;
while( !DONE )
{
  tmp = tmp / 2;
ctr[i] ++;
if( tmp == 0 )  DONE = TRUE;
}
  }
printf("# conversion table look-up for RNS with  { %d, %d, %d } \n", mi[1],
mi[2], mi[3]);
printf(".i  %d \n", ctr[1] + ctr[2] + ctr[3]);
printf(".o  %d \n", ctr[4]);
printf(".phase ");
for(i = 1; i <= ctr[4]; i++) printf("1");
printf("\n \n ");
/* COMPLETION OF BIT CALCULATION PORTION OF PROGRAM */

/* Calculate the bit fields required and output   */
for( i = 0; i < M; i++ )
   {
for( k = 1; k <= 4; k++ )
{
DONE = FALSE;
raise_it = ctr[k];
tmp = i;

while( !DONE )
{
tmp = tmp % mi[k];
raise_it = raise_it - 1;
tmp_div = 1;

for(j = 1; j <= raise_it; j++)
tmp_div = tmp_div * 2;
digit = tmp / tmp_div;
tmp = tmp % tmp_div;
if(digit != 0) printf("1");
else  printf("0");
if( raise_it == 0 )  DONE = TRUE;
}
printf(" ");
}
printf("\n");
   }
printf(".e");
}
```

2. C Code for Quotient Table Generation

```
/*********************************************************
*    PROGRAM NAME:      qlugen.c                          *
*    FUNCTION: Generates minterm equations in the format *
*               required by the VLSI tool "espresso" for  *
*               minimization and then implementation as   *
*               PLA for an RNS-QD quotient look-up table. *
*    VERSION:   1.4                                       *
*    AUTHOR:    David E. Gilbert                          *
*    Date Last Changed: 10 Sep 1991                       *
*********************************************************/
#include <stdio.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

main(argc, argv)
int argc;
char **argv;
{
int mi[5], M, i, j, k, ctr[5], biggest, tmp, DONE;
int enter, digit, tmp_div, raise_it;

M=1;
for(i=1; i< argc; i++) { mi[i]=atoi(argv[i]);
                         M = M*mi[i];        }
/* Find the largest moduli                    */
biggest = 0;
for(i = 1; i < argc; i++)
  {
       if(mi[i] > biggest)      biggest = mi[i];
  }
mi[4] = M / biggest;

/* CALCULATE THE POWERS OF 2 NEEDED FOR NUMBERS OF BITS NEEDED */
for(i = 1; i <= 4; i++)
   {     ctr[i] = 0;
         tmp = mi[i];
         DONE = FALSE;
         while( !DONE )
         {     tmp = tmp / 2;
               ctr[i] ++;
               if( tmp == 0 )  DONE = TRUE;    }     }/*end of while */

printf("# conversion table look-up for RNS with  { %d, %d, %d } \n", mi[1],
mi[2], mi[3]);
printf(".i  %d \n", ctr[1] + ctr[2] + ctr[3]);
printf(".o  %d \n", ctr[4]);
```

```
printf(".phase  ");
for(i = 1; i <= ctr[4]; i++) printf("1");
printf("\n \n ");
/* COMPLETION OF BIT CALCULATION PORTION OF PROGRAM */


/* Calculate the bit fields required and output   */
mi[4] = biggest; /* This is a temporary fix and should be corrected */
for( i = 0; i < M; i++ )
   {
      for( k = 1; k <= argc; k++ )
      {
           DONE = FALSE;
           raise_it = ctr[k];
           tmp = i;
           enter = 1;
           while( !DONE )
           {
               if(k == argc )
               {
                   if( enter == 1)  tmp = tmp / mi[k];
                   else tmp = tmp % mi[k];
                   enter ++;
                }
               else    tmp = tmp % mi[k];
               raise_it = raise_it - 1;
               tmp_div = 1;

               for(j = 1; j <= raise_it; j++)
                    tmp_div = tmp_div * 2;

               digit = tmp / tmp_div;
               tmp = tmp % tmp_div;
               if(digit != 0) printf("1");
               else           printf("0");
               if( raise_it == 0 )       DONE = TRUE;
             }
            printf(" ");
       }
       printf("\n");

   }
printf(".e");
}
```

APPENDIX B: SAMPLE EQUATIONS

1. Output from QLUGEN.C

```
# conversion table look-up for RNS with  { 2, 3, 5 }
.i  7
.o  3
.phase 111
00 00 000 000
01 01 001 000
00 10 010 000
01 00 011 000
00 01 100 000
01 10 000 001
00 00 001 001
01 01 010 001
00 10 011 001
01 00 100 001
00 01 000 010
01 10 001 010
00 00 010 010
01 01 011 010
00 10 100 010
01 00 000 011
00 01 001 011
01 10 010 011
00 00 011 011
01 01 100 011
00 10 000 100
01 00 001 100
00 01 010 100
01 10 011 100
00 00 100 100
01 01 000 101
00 10 001 101
01 00 010 101
00 01 011 101
01 10 100 101
.e
```

2. Reduced Equations from *Espresso*

The following is a listing of the reduced set of equations from the original moduli set {3, 5, 7} after being processed by *espresso*. These are the equations that would be used to generate the quotient look–up table PLA with the VLSI tool *mpla*.

```
# conversion table look-up for RNS with  { 2, 3, 5 }
.i 7
.o 3
#.phase 111
.p 20
0010100 010
0110100 100
0101011 010
0000100 100
0110011 100
0101100 011
0110001 010
0101000 100
000101- 100
0100010 100
0110010 011
0100001 100
000100- 010
001000- 100
0100000 010
000001- 010
000-0-1 001
00-00-1 001
01-0-00 001
010-0-0 001
.e
```

APPENDIX C: SAMPLE RNL FILES

1. Sample RNL Execution File

The following is a listing of an execution file to simulate the RNS-QD system {2, 3, 5}.

```
;  The name of this control file for rnl is: qlu2.l
;  This is the control file for simulation on a PLA for quotient look-up

;  LOAD STANDARD LIBRARY ROUTINES
(load "uwstd.l")
(load "uwsim.l")

;  FILE WHICH WILL LOG THE RESULTS
(log-file "qlu2.rlog")

;  READ IN THE BINARY NETWORK FILE
(read-network "qlu2")
(sim-init)

;  DEFINE THE TIME SCALE FOR SIMULATION
(setq incr 10)

;  DEFINE INPUT VECTOR IF ANY, standard STYLE
(defvec '(bit status input_1 input_2 input_3 input_4 input_5 input_6 input_7))
(defvec '(bit output output_1 output_2 output_3))

;  DEFINE INPUT VECTOR IF ANY, SINGLE INDEX STYLE

;  DEFINE INPUT VECTOR IF ANY, double index STYLE

;  STANDARD REPORT FORMAT DEFINITION.
(def-report '("response = " clka clkabar (vec output) (vec status)))

;  PLOTFILE SPECIFIED
openplot "qlu2.beh"

;  LOGIC ANALYZER STYLE OUTPUT FORMAT SELECTION.
(setq lanalyze t)
(wr-format)

;  GLITCH DETECTOR SELECTION.

;  NODE TRANSIENTS REPORT DEFINITION.
(chflag '( output_1 output_2 output_3))

;  TRIGGER CONDITION SET-UP
```

```
;  ADDITIONAL SIMULATION SET-UP COMMAND LINES.
(printf "The simulation starts now...\n")

;  SPECIFICATION OF A TIME/BASENAME FILE FOR INCLUSION.
(load "qlu2.time")

;  ADDITIONAL WRAP-UP COMMAND LINES.
(printf "simulation completed ... check file  *.rlog for results. \n")
exit

;  GEN-CONTROL COMPLETED.
```

2. Sample RNL Simulation Output

```
; 62 nodes, transistors: enh=157 intrinsic=0 p-chan=30 dep=0 low-power=0 pullup=0 resistor=0

; Report format of logic analyzer style output
 time  clka  clkabar  output  status

The simulation starts now...
output_1 = 1 @ 0
output_2 = 1 @ 0
output_3 = 1 @ 0
1  0 0 111 0000000
output_1 = 0 @ 0.5
2  1 0 011 0000000
output_3 = 0 @ 0.2
output_2 = 0 @ 0.2
output_2 = 1 @ 0.9
output_3 = 1 @ 0.9
3  0 0 011 0010101
output_1 = 1 @ 0.2
4  1 0 111 0010101
output_3 = 0 @ 0.4
output_2 = 0 @ 0.4
output_1 = 0 @ 0.7
5  0 0 000 0001001
output_3 = 1 @ 0.8
output_2 = 1 @ 0.9
6  1 0 011 0001001
output_2 = 0 @ 0.6
7  0 0 001 0000100
8  1 0 001 0000100
output_3 = 0 @ 0.3
output_2 = 1 @ 0.4
9  0 0 010 0001000
output_2 = 0 @ 0.5
output_1 = 1 @ 0.9
10  1 0 100 0001000
output_2 = 1 @ 0.8
11  0 0 110 1000100
```

```
output_1 = 0 @ 0.6
output_2 = 0 @ 0.6
output_2 = 1 @ 0.9
output_1 = 1 @ 0.9
12  1 0 110 1000100
output_1 = 0 @ 0.7
output_2 = 0 @ 0.9
13  0 0 000 0000101
output_3 = 1 @ 0.9
output_1 = 1 @ 0.9
14  1 0 101 0000101
15  0 0 101 0101000
output_1 = 0 @ 0.5
16  1 0 001 0101000
output_3 = 0 @ 0.3
output_3 = 1 @ 0.9
17  0 0 001 0010000
18  1 0 001 0010000
output_1 = 1 @ 0.9
19  0 0 101 0011001
output_2 = 1 @ 0.2
20  1 0 111 0011001
output_3 = 0 @ 0.4
output_1 = 0 @ 0.6
21  0 0 010 0010100
output_2 = 0 @ 0.6
22  1 0 000 0010100
output_3 = 1 @ 0.4
output_2 = 1 @ 0.4
23  0 0 011 0000001
output_2 = 0 @ 0.5
output_3 = 0 @ 0.6
output_3 = 1 @ 0.8
output_2 = 1 @ 0.9
24  1 0 011 0000001
output_1 = 1 @ 0.9
25  0 0 111 0010001
26  1 0 111 0010001
output_3 = 0 @ 0.4
output_2 = 0 @ 0.4
output_2 = 1 @ 0.9
27  0 0 110 0011000
28  1 0 110 0011000
output_1 = 0 @ 0.3
output_2 = 0 @ 0.3
output_3 = 1 @ 0.4
output_1 = 1 @ 0.4
29  0 0 101 0100101
output_1 = 0 @ 0.6
output_3 = 0 @ 0.6
output_3 = 1 @ 0.8
```

```
output_2 = 1 @ 0.9
output_1 = 1 @ 0.9
30  1 0 111 0100101
output_2 = 0 @ 0.5
output_1 = 0 @ 0.7
31  0 0 001 1001001
output_2 = 1 @ 0.9
output_1 = 1 @ 0.9
32  1 0 111 1001001
output_2 = 0 @ 0.6
33  0 0 101 1001001
simulation completed ... check file  *.rlog for results.
```

REFERENCES

1. Soderstrand, M. A.,and others, *Residue Number System Arithmetic*, Addison–Wesley Publishing Company, 1986.

2. Knuth, D. E., *The Art of Computer Programming*, Addison–Wesley Publishing Company, 1969.

3. Soderstrand, M. A., "A New Hardware Implementation of Modulo Adders for Residue Number Systems," *Proceedings of the 26th Midwest Symposium on Circuits and Systems*, pp. 412-415, 1983.

4. Taylor, F. J., "Large Moduli Multipliers for Signal Processing," *IEEE Transactions on Circuits and Systems*, vol. CAS–28, pp. 731-736, July, 1981.

5. Taylor, F. J. and Ramnarayanan, A. S., "An Efficient Residue–to–Decimal Converter," *IEEE Transactions on Circuits and Systems*, Vol. CAS–28, pp. 1164-1169, December, 1981.

6. Cheney, P. W., "A Digital Correlator Based on the Residue Number System," *IRE Transactions on Electronic Computing*, vol. EC–11, pp. 63-70, March, 1961.

7. Golomb, S. W., *Shift Register Sequences*, Aegean Park Press, Revised Edition, pp. 225-228, 1982.

8. Lin. S. and Costello, D. J., *Error Control Coding: Fundamentals and Applications*, Prentice–Hall Inc., 1983.

9. Yang, C., Lu, H.–C., and Gilbert, D. E., "An Investigation into the Implementation Costs of Residue and High Radix Arithmetic," paper presented at the 21st International Symposium for Multiple-Valued Logic , Victoria, Canada, 25 May 1991.

10. Watson, R. W. and Hastings, C. W., "Self–Checked Computation Using Residue Arithmetic," *Proceedings of the IEEE*, vol. 54, pp. 1920-1931, December, 1966.

11. Yau, S. S. and Liu, Y. C., "Error Correction in Redundant Residue Number Systems," *IEEE Transactions on Computing*, vol. C–22, pp. 5-11, January, 1973.

12. Weste, N. H. E. and Eshraghian, K., *Principles of CMOS VLSI Design*, Addison–Wesley Publishing Co., pp. , October, 1985.

13. Wakerly, J. *Digital Design Principles and Practices*, Prentice Hall Inc, pp. 556-564, 1990.

14. Roth, C. H. Jr., *Fundamentals of Logic Design*, West Publishing Co., 1985.

15. Scott, W. S. and others, "1986 VLSI Tools: Still More Works by the Original Artists," Report No. UCB/CSD, University of California, pp. 91 - 142, December 1985.

16. Bayoumi, M. A., Jullien, G. A., and Miller, W. C., "Models for VLSI Implementation of Residue Number System Arithmetic Modules," *Proceedings of the 6th Symposium on Computer Arithmetic*, pp. 174-182, 1983.

17. Soderstrand, M. A., Vernia, C., and Chang, J., "An Improved Residue Number System Digital–to–Analog Converter," *IEEE Transactions on Circuits and Systems*, vol. CAS–30, pp. 903-907, December, 1983.

18. Lipschutz, S., *Finite Mathematics*, McGraw–Hill Book Company, pp. 58-65, 1966.

19. Hennessey, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc, pp. 251-341, 1990.

20. Telephone conversation between Barry Karsch, Naval Air Development Center, Warminster PA., and the author, 12 August 1991.

21. NW Laboratory for Integrated Systems, Report #88-09-01, *VLSI Design Tools Manual*, Department of Computer Science, Section 6.3 pp. 1-32, 1 September 1988.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Professor Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Lt. David E. Gilbert 244 Overholt Drive Virginia Beach, VA 23462	1
7.	Dr. George Abraham, Code 1005 Office of Research and Technology Naval Research Laboratories 4555 Overlook Ave., N.W. Washington, DC 20375	1

8. Dr. Robert Williams 1
Naval Air Development Center, Code 5005
Warminster, PA 18974-5000

9. Dr. James Gault 1
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709

10. C. Lee Giles 1
AFOSR, Bldg. 410
Bolling, AFB, DC 20332

11. Dr. Andre van Tilborg 1
Office of Naval Research
Code 1133
800 N. Quincy Str.
Arlington, VA 22217-5000

12. Dr. Clifford Lau 1
Office of Naval Research
1030 E. Green Str.
Pasadena, CA 91106-2485